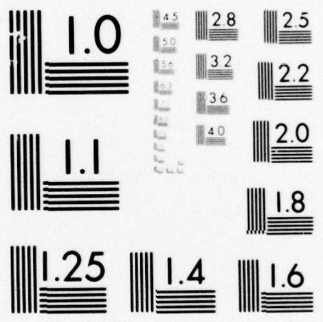
AD-A037 467 UNIVERSITY OF SOUTHERN CALIFORNIA LOS ANGELES F/G 9/2
RECENT DEVELOPMENTS IN SOFTWARE FAULT TOLERANCE THROUGH PROGRAM--ETC(U)
1976 K H KIM F44620-71-C-0061
UNCLASSIFIED AFOSR-TR-77-0148 NL

| OF |
AD
A037467

END
DATE
FILMED
4-77


1.0
4.5
5.0
5.6
6.3
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.1
1.8
1.25
1.4
1.6

Proceedings of the 10th Hawaii Int'l Conference on System Sciences, Jan. 6-7, 1977.

2

RECENT DEVELOPMENTS IN SOFTWARE FAULT TOLERANCE THROUGH PROGRAM REDUNDANCY[†]

K. H. Kim
University of Southern California
Los Angeles, California

C. V. Ramamoorthy
University of California,
Berkeley, California

ABSTRACT

This paper briefly reviews some of the recent developments in software fault tolerance through program redundancy. Language constructs, system architectures and design methodologies developed or proposed for programs containing redundancy for error tolerance, are reviewed.

1. INTRODUCTION

Production of large-scale error-free programs is a difficult task. Tolerance of software faults means that, during program execution, the damaging effects of residual design errors are contained and acceptable results are continuously generated for the environment. In recent years the possibility of using program redundancy for the purpose of providing software fault tolerance has become a subject of growing interest [1-27]. A program containing redundancy for the tolerance of software and/or hardware faults is called a fault-tolerant program [6].

A fault-tolerant program designed for tolerating residual design errors normally contains multiple routines that can compute the same function. If these routines use distinct algorithms, it is less likely that the same design error will exist in all of the routines that compute the function. The use of program redundancy for software fault tolerance is motivated by the present situation in software engineering:
(1) In spite of recent advances in program validation and design methodology, large-scale programs are often put into operation with residual design errors.
(2) Computing systems are increasingly employed in critical environments where the cost of failure is extremely high.
(3) Maintenance and modification of operational programs are practical necessities and often introduce new errors.
(4) Hardware reliability and economy have been significantly improved in recent years, substantially increasing the freedom in using extra hardware.

Recent developments in fault-tolerant programming have taken place in several areas: (1) language constructs for structuring fault-tolerant programs, (2) system architectures for processing fault-tolerant programs, (3) design methodologies for fault-tolerant programs. This paper is a brief overview of some of these developments.

2. CHARACTERISTICS OF FAULT-TOLERANT PROGRAMMING

Program redundancy can be used in many different forms in achieving fault-tolerance. In a well-structured system (e.g., hierarchically structured as discussed in [19, 21, 22]), the same schemes for incorporating program redundancy can be used for both hardware and software fault tolerance. In [12], Horning et al. identified three major types of program redundancy useful for achieving error tolerance: (1) acceptance tests which check the reasonableness or acceptability of intermediate results during program execution, (2) alternate routines which are invoked to compute the same objective function in different ways when the result produced by the primary routine is rejected, and (3) recovery routines which, when invoked on rejection of an intermediate result by an acceptance test, restore the system state to a "legitimate" state (where a retry or execution of the rest of the program can start).

The use of program redundancy is not without cost, however. Fault-tolerant programs have generally a high development cost and a higher processing cost than conventional programs. One way of reducing the development cost is to develop methodologies and tools for systematic incorporation of redundancy in a well-structured form. Good structuring of a fault-tolerant program is particularly important since the use of program redundancy increases the program size.

The high processing cost of fault-tolerant programs is due to the increased processor time and increased storage space required for exe-

† This work was supported in part by the Joint Service Electronics Program under Air Force Contract F44620-76-c-0061.

1473

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

(19) **REPORT DOCUMENTATION PAGE**		READ INSTRUCTIONS BEFORE COMPLETING FORM
(18) 1. REPORT NUMBER AFOSR - TR - 77 - 0148	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
(6) 4. TITLE (and Subtitle) RECENT DEVELOPMENTS IN SOFTWARE FAULT TOLERANCE THROUGH PROGRAM REDUNDANCY,		(9) 5. TYPE OF REPORT & PERIOD COVERED INTERIM rept.,
		6. PERFORMING ORG. REPORT NUMBER
(10) 7. AUTHOR(s) Kim, K.H. /Kim Ramamoorthy, C.V.		(15) 8. CONTRACT OR GRANT NUMBER(s) F44620-71-C-0061 new
9. PERFORMING ORGANIZATION NAME AND ADDRESS University of Southern California Los Angeles, CA		(16) 10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 61102F 2305/A9 (17)
11. CONTROLLING OFFICE NAME AND ADDRESS AFOSR/NE Bolling AFB, DC 20332 (12) 7p.		(11) 12. REPORT DATE 1976
		13. NUMBER OF PAGES 6
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

Proseedings of 10th Hawaii Int'l Confe rence on System Sciences Jan 6-7 1977

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This paper briefly reviews some of the recent developments in software fault tolerance through program redundancy. Language constructs, system architectures and design methodologies developed or proposed for programs containing redundancy for error tolerance, are reviewed.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED 361 550

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered) cuting redundant program components. For example, to provide a recovery capability requires processor time and storage space for recording and retaining some histories of computation (i.e., saving of system states at various points in program execution). Execution of acceptance tests in fault-tolerant programs can also be a factor contributing to a large execution time [4,6]. System architectures reviewed in section 4 aim at the reduction of execution time and/or storage space.

3. LANGUAGE CONSTRUCTS FOR FAULT-TOLERANT PROGRAMMING

A language construct developed by Horning et al., called a recovery block or fault-tolerant block, allows incorporation of program redundancy into a block-structured program in a well-structured manner [2,12,21]. A fault-tolerant block has the following structure: ensure T by O_1 else-by O_2 ... else-by O_n else-error, where T denotes the acceptance test, O_1 the primary object block and O_k $(1<k \leq n)$ the alternate object blocks. All the object blocks in a fault-tolerant block F compute the same or approximately the same objective function. The acceptance test T is executed on exit from an object block to confirm that the object block has performed acceptably. The execution of an acceptance test results in either an acceptance (i.e., confirmation) or a rejection. If accepted, control exits from the fault-tolerant block. If the result produced by an object block is rejected, the next alternate object block is entered. After the alternate object block finishes its computation, the acceptance test is repeated.

The following aspects of this scheme should be noted: (1) the primary or alternate object blocks can contain, nested within themselves, further fault-tolerant blocks; (2) execution of the acceptance test may require reference to both the original and the modified values of variables non-local to the object block; and (3) it is not necessary that every block in a block-structured fault-tolerant program be a fault-tolerant block. This flexibility is particularly important since it allows incorporation of program redundancy only where desired.

Before an alternate object block is entered, the system state is restored to the state that existed just before entry to the primary object block. Each variable that was assigned a new value by the rejected execution is restored to the original value of the variable. The system automatically performs this "assignment reversal". A state vector that contains the values of all the variables (that may be changed by the object blocks) is saved on entry to a fault-tolerant block. If assignment reversal is later necessary, this state vector is used to restore the earlier variable values. Note that the programmer is relieved from specifying this "standard" recovery action.

However, certain components of a system cannot be restored in such a manner. For example, operations involving file access, accounting operations, or operations dependent upon a real-time clock cannot be reversed by the method used for automatic reversal of assignments. Horning et al. proposed a language construct, called a recoverable procedure, for the specification of non-standard recovery operations [12]. A recoverable procedure allows the programmer to specify (1) the opterations of saving information that may be required for non-standard recovery, (2) the normal computation, and (3) the non-standard recovery operations themselves.

Checking the validity of input to a program module is a common programming practice especially in real-time applications [11] and two approaches to the incorporation of the input check into a fault-tolerant block are compared in [17]. One approach is to place before a fault-tolerant block F an "empty" fault-tolerant block F^e which contains nothing but an acceptance test T^e which checks the validity of input to F. The other approach is to extend the acceptance test to consist of two parts, the input acceptance test T^I and the result acceptance test T^R. The syntactic structure thus becomes: ensure on-entry T^I on-exit T^R by O_1 else-by O_2 ... else-error. The input acceptance test T^I is executed "before" the primary object block O_1. Both approaches may be used in structuring a fault-tolerant program.

If a loop in a fault-tolerant program contains a fault-tolerant block as its body, then the acceptance test is entered at every iteration to confirm the result. Sometimes it is desirable to invoke the validation process less frequently than at every iteration. For flexible invocation of the acceptance test, an extension of the fault-tolerant block was proposed to include a predicate so that when the predicate is "false" on entry to the fault-tolerant block, only the primary object block is executed without invoking the execution of the acceptance test [17]. A syntactic structure of the extended fault-tolerant block is: ensure when < logical expression> that on-entry T^I on-exit T^R by O_1 else-by O_2 ... else-error.

For real-time applications, a watchdog timer which is set on entry to an object block and signals an error if a preset time limit is reached, can be useful [4,11,12,18]. Elmendorf discussed a set of (assembly) language primitives for controlling both concurrent execution of several alternates and multiple executions of the same routine [6].

Exception handling is a generic term referring to detecting and responding to abnormal or undesired events [10, 13, 19]. Thus the block-structured fault-tolerant programming discussed above can be viewed as a highly structured form of exception handling in which error detection normally occurs on completion of an object block and recovery generally involves complete cleanup of the contaminated part of the system by a backup to an earlier state. In contrast, most approaches to exception handling aim at immediate error detection (i.e., detection during execution of a statement/instruction) and "efficient patching" of the system from an undesired state into a legitimate state (e.g., PL/1 ON-construct). Using the language constructs designed to support such exception handling, error detection and recovery can be structured in a flexible and unrestricted manner (e.g., non-nested structure). However, they impose a greater burden on the programmer. It seems possible to realize the immediate error detection and efficient system patching effect to a large extent by extensive use of recoverable procedures in the fault-tolerant block scheme.

When a set of parallel processes interact, each capable of self-checking and recovery, an error in one process may affect a different process with which it has interacted [8, 21, 23, 24]. For instance, if process A has produced data for process B and this data is later determined to be incorrect by an acceptance test in process A, then both process A and process B may have to be restored to an earlier state. Russell and Bredt showed that explicit error handling could lead to complicated program structure even in the simple case of a producer-consumer system [23]. Randell recognized that uncontrolled use of process communication along with error detection and recovery could lead to a disastrous avalanche of backup activity called the domino effect [21]. In order to control the domino effect Randell proposed to restrict the interactions of parallel processes to take place inside conversations [21]. A conversation is a fault-tolerant block which spans two or more processes and creates a "boundary" which process interactions may not cross. On entry to a conversation, the state of each process is saved and if any process fails its own acceptance test at the end of the conversation, all the processes are backed up to the beginning of the conversation. Power of conversations as tools for structured fault-tolerant parallel programming remains to be conclusively demonstrated.

4. SYSTEM ARCHITECTURES FOR PROCESSING FAULT-TOLERANT PROGRAMS

Since the processing of fault-tolerant programs involves saving of state vectors and examination of saved state vectors (for validation and recovery), the processing cost can be reduced by saving state vectors in a compact form. It is useful to save only the differences between successive state vectors, rather than to save the entire state vector on entry to each fault-tolerant block. The processing cost can also be reduced by noting that the varibles local to the object block are irrelevant to the recovery and in many cases, only a small subset of the non-local variables are modified by the object block. To exploit these properties, Horning et al. developed a scheme called the recovery cache which saves state vectors in a compact form [2, 12]. The essence of this scheme is to record the original value of each non-local variable together with its logical address in a table right before the variable is modified for the first time in a new block.

Validation of results and saving of state vectors contribute to an increase in program execution time. When any non-trivial validation is employed or large numbers of non-local variables are modified during execution of object blocks, the program execution time even in the case of normal fault-free operation could very well exceed the tolerable limit in real-time applications [4, 6]. (It is indeed expected that useful acceptance tests may frequently be quite complex.) As a possible solution, an approach of overlapping object block execution with the validation and state vector saving was proposed in [14]. This parallel execution approach requires a multiprocessor system in which (1) histories of the main-stream computation are kept in a compact form, and (2) interference between processors, particularly storage conflicts, is negligible or absent. A multiprocessor system architecture that satisfactorily meets these requirements by employing a novel memory organization, named a duplex memory, was sketched in [14] and fully described in [15]. The duplex memory also offers compact storage for state vectors but in a different way than the recovery cache. Performance evaluation of this system based on a stochastic model confirmed high effectiveness of the parallel system in reducing the execution time increase due to validation and state vector saving [16].

There are other systems already developed or under development which support the structured use of program redundancy for error detection and recovery, e.g., ESS No. 2 [4], System 250 [22], PRIME [7], HYDRA [25], etc. These systems do not employ special hardware components unlike the systems discussed above. EES No. 2 uses extensive redundancy in data structures and periodically calls in an audit program (i.e., a collection of validation tests) to check the consistency of the data structures and the integrity of data. Thus audit programs in this system are loosely coupled with application programs. Yau et al. proposed a software subsystem, called the system monitor, that interfaces with the operating system on one side and with application programs on the other side [27]. The system monitor runs in a more privileged protection domain than the application programs and is responsible for safe execution of user-supplied error detection and recovery codes designed to ensure acceptable performance of the application programs. In System 250, the software subsystem is hierarchically structured and each level contains its own set of software checks (for error detection) and recovery procedures. If the recovery function at a level fails, then the recovery at a lower level (i.e., a level closer to the system nucleus) takes over, analogous to a hierarchy of recovery functions built in multi-level nested fault-tolerant blocks. PRIME is a multiprocessor time-sharing system and uses the technique of dynamic decision verification which means that, every time the operating system running on one processor makes a decision there is a followup consistency check that uses a different algorithm and runs on a different processor. HYDRA is a kernel (nucleus) operating system for C.mmp multiprocessor system and is composed of modules each capable of input check, acceptance test and error report. For important data structures, their descriptors (i.e., type and size information) are stored in redundant forms.

5. OTHER DEVELOPMENTS AND UNRESOLVED ASPECTS

Progress has also been made in other aspects of fault-tolerant programming. Some efforts have been made to develop effective design methodologies for fault-tolerant programs. Anderson proposed a proof-guided methodology for designing acceptance tests [1]. He asserted that the use of acceptance tests should simplify proofs of adequacy (weaker than correctness), and in turn, the attempts to construct proofs should provide indications of where acceptance tests could usefully be employed. Kopetz argued for the usefulness of an alternate object block which is designed to provide minimally acceptable service to the rest of the program [18]. He then proposed, for efficient implementation and debugging, to start program development with the construction of acceptance tests and alternate object blocks performing minimally acceptable functions and then to proceed to the construction of primary object blocks. Parnas and Würges identified several basic types of exceptional conditions and discussed methods of structuring inter-level exception handling in hierarchically organized systems [19]. Russell was concerned with fault-tolerant inter-process communication that is both domino-free and efficient [24]. He considered systems of parallel processes which were constructed by using primitives such as MARK for saving system states, RESTORE for restoring the system state and PURGE for discarding no longer useful system state information, and then studied conditions under which systems using these primitives can be guaranteed not to be subject to any possible domino effect. Gilb reported that the approach of dual coding (i.e., developing two different versions of a program with the same specification by two independent programmers or the same programmer working in different languages) had been highly cost-effective in many software projects [9]. Successful application of this approach to obtaining alternates in fault-tolerant programs was also reported. Some error detection and recovery techniques used in practice were reviewed by Yau and Cheung [26] and also in [20].

Denning summarized a broad range of principles of structuring operating systems to make unauthorized actions impossible, to confine errors to the immediate contexts of their occurrences, and to detect and correct damage before it spreads [5]. He also stressed the need for considerable hardware support in efficient implementation of those principles.

Fault-tolerant programming is still in its infancy. Only limited experience with fault-tolerant programs has been obtained, although highly optimistic projections have come out of the limited experience [2, 4, 9, 11, 18]. Much more work is needed in many areas, among them the following:
(1) the development of system aids which are useful at other phases of the program development cycle - specification, validation, analysis, maintenance;
(2) the development of additional language constructs, particularly for fault-tolerant parallel programming;
(3) the development of design methodologies for fault-tolerant programs, i.e., for acceptance tests and alternates;
(4) the establishment of a theoretical foundation for fault-tolerant programming.

Further experience with fault-tolerant programming will provide additional insight into the directions of future work and into the full potential of fault-tolerant programmming.

REFERENCES

1. Anderson, T., "Provably safe programs," Tech. Rept. No. 7, Computing Lab., Univ. of Newcastle upon Tyne, February 1975.

2. Anderson, T. and Kerr, R., "Recovery blocks in action: a system supporting high reliability" Proc. 2nd Int'l Conf. on Software Engineering, 1976, pp. 447-457.

3. Avizienis, A., "Fault-tolerance and fault-intolerance: complementary approaches to reliable computing," Proc. 1975 Int'l Conf. on Reliable Computing, pp. 458-464.

4. Connet, J.R. et al, "Software defenses in real-time control systems," Digest of the 1972 Int'l Symp. on Fault-Tolerant Computing, pp. 94-99.

5. Denning, P., "Fault-tolerant operating systems," Tech. Rept. CSD TR-175, Computer Science Dept., Purdue Univ., Apr. 1976 (to be published in Computing Surveys).

6. Elmendorf, W.R., "Fault-tolerant programming," Digest of the 1972 Int'l Symp. on Fault-Tolerant Computing, pp. 79-83.

7. Fabry, R.S., "Dynamic verification of operating system decisions," Comm. ACM, November 1973, pp. 659-668.

8. Gerstmann, H., Diel, H. and Witzel, W., "The reliability of programming systems," Lecture Notes in Comp. Sci., vol. 23, Springer-Verlag, 1974, pp. 87-113.

9. Gilb, T., "Parallel programming," Datamation, October 1974, pp. 160-161.

10. Goodenough, J.B., "Exception-handling: issues and a proposed notation," Comm. ACM, December 1975, pp. 683-696.

11. Hecht, H., "Fault-tolerant software for spacecraft applications," Tech. Rept. SAMSO-76-40, Aerospace Corp., Dec. 1975.

12. Horning, J.J. et al, "A program structure for error detection and recovery," Lecture Notes in Comp. Sci., vol. 16, Springer-Verlag, 1974, pp. 171-187.

13. IBM Corp., 'IBM System/360 operating system PL/1 (F) language reference manual,' Order No. GC28-8201-4, Dec. 1972.

14. Kim, K.H. and Ramamoorthy, C.V., "Failure-tolerant parallel programming and its supporting system architecture," Proc. AFIPS Nat'l Comp. Conf., 1976, pp. 413-423.

15. Kim, K.H., "A parallel system processing fault-tolerant programs - I. system architecture," submitted to IEEE Trans. on Computers.

16. Kim, K.H., Jenson, M.J. and Olumi, M., "A parallel system processing fault-tolerant programs - II. performance evaluation," submitted to IEEE Trans. on Computers.

17. Kim, K.H., Russell, D.L. and Jenson, M.J., "Language tools for fault-tolerant programming," submitted for publication.

18. Kopetz, H., "Software redundancy in real-time systems," Proc. IFIP Congress 1974, pp. 182-186.

19. Parnas, D.L. and Würges, H., "Response to undesired events in software systems," Proc. 2nd Int'l Conf. on Software Engr., 1976, pp. 437-446.

20. Ramamoorthy, C.V., Cheung, R.C. and Kim, K.H., "Reliability and integrity of large computer programs," Lecture Notes in Comp. Sci., Vol. 12, Springer-Verlag, 1974, pp. 86-161.

21. Randell, B., "System structure for software fault tolerance," IEEE Trans. on Software Engr., June 1975, pp. 220-232.

22. Repton, C.S., "Reliability assurance for System 250 a reliable, real-time control system," Proc. Int'l Computer Communication Conf., 1972, pp. 297-305.

23. Russell, D.L. and Bredt, T.H., "Error resynchronization in producer-consumer systems," Proc. 5th Symp. on Operating System Principles, Nov. 1975, pp. 106-113.

24. Russell, D.L., "State restoration among communicating processes," Tech. Rept. No. 112, Digital System Laboratory, Stanford University, June 1976.

25. Wulf, W.A., "Reliable hardware-software architecture," Proc. 1975 Int'l Conf. on Reliable Software, pp. 122-130.

26. Yau, S.S. and Cheung, R.C., "Design of self-checking software," Proc. Int'l Conf. on Reliable Software, Apr. 1975, pp. 450-457.

27. Yau, S.S., Cheung, R.C. and Cochrane, D.C., "An approach to error-resistant software design," Proc. 2nd Int'l Conf. on Software Engineering, 1976, pp. 429-436.

K.H. Kim was born in Korea in 1947. He received the B.S. degree in electrical engineering from Seoul National University, Seoul, Korea, the M.A. degree in computer Science from the University of Texas, Austin, and the Ph.D. degree in electrical engineering and computer science from the University of California, Berkeley, in 1969, 1972 and 1974, respectively.

From 1969 to 1971 he was an officer in the Korean Army. He worked as a Research Assistant in the Electronics Research Center of the University of Texas at Austin from 1971 to 1972, and in the Electronics Research Laboratory of the University of California, Berkeley, form 1972 to 1974. During the summer of 1974, he worked as an Acting Instructor in the Computer Science Division of the University of California, Berkeley. Since January 1975, he has been an Assistant Professor of Electrical Engineering and Computer Science at the University of Southern California, Los Angeles.

C.V. Ramamoorthy received the undergraduate degrees in physics and technology from the University of Madras, India, the M.S. degree and the professional degree of Mechanical Engineer, both from the University of California, Berkeley, and the M.A. and Ph.D. degrees in applied mathematics and computer theory from Harvard University, Cambridge, Mass.

He was associated with Honeywell's Electronic Data Processing Division, Waltham, Mass., for over eleven years, most recently as Senior Staff Scientist. He was a Professor in the Departments of Computer Science and Electrical Engineering at the University of Texas, Austin. He is currently a Professor in the Department of Electrical Engineering and Computer Science at the University of California, Berkeley.